

## CHAPTER II

### Syntax and Semantics

An *assertive program* is a computer program that not only gives the instructions to be followed by the computer when it executes the program, but also *asserts*, in the *specification* portions of the program, what is to be accomplished when the program is executed. An assertive program that satisfies its specification in every possible execution is said to be *valid* (or *correct*). The syntax and semantics of a related language (called “RESOLVE”) for expressing assertive programs have been discussed in earlier work of the Reusable Software Research Group (RSRG) at The Ohio State University [44, 49, 22, 25, 16, 45].

Krone [25] has already established a method for proving correctness for assertive programs, the major contributions being rules for handling module-level constructs and interactions among modules. The rules governing procedure correctness followed the tradition of Hoare logic, which we have claimed is not as natural as our proposed method. We therefore replace the rules for procedure bodies with new rules that formalize the indexed method—the more natural method of reasoning presented in Section 1.5.2. We are not proposing changes to the rules for modules. The only change we are proposing to the rules for procedure declaration is a single new rule that provides a bridge between Krone’s rules and the indexed method.

Krone’s system reduces assertive programs to assertions to be proved in ordinary logic. Furthermore, following Ernst et al. [12, p. 149], her system “views the intermediate objects in the reduction process as extended programs, thereby making verification a much less abstruse process. Treating logical assertions as commands appeals strongly to a programmer’s intuition.” Morgan [36, p. 403] proposed a similar programming-language extension to aid program refinement. He extended Dijkstra’s programming language with *specification statements*.

The goal is to improve the *development* of programs, making it closer to manipulations within a single calculus. The extension does this by providing one semantic framework for specifications and programs alike:

Developments begin with a program (a single specification statement) and end with a program (in the executable language).

The approaches of Ernst et al. and Morgan are similar in that they both establish a single syntax and semantics for programs-with-specifications, i.e., assertive programs. The main difference is that Morgan focuses on developing an executable assertive program given an assertion serving as the specification, while Ernst et al. focus on deriving an assertion from a given assertive program to verify its correctness. We follow the approach of Ernst et al. and Morgan, giving a single syntax and semantics that encompasses all of the following:

1. executable assertive programs,
2. the final, derived, logical assertion, and
3. all the intermediate forms.

More specifically, we follow Ernst et al., focusing on the derivation of assertions from executable assertive programs for the purpose of verification.

The indexed method focuses on procedure bodies—bodies that do not contain declarations. In particular, these procedure bodies do not contain variable, procedure, or module declarations. It will be convenient to redefine our terminology to match this new restricted focus. Henceforth, we use the word “program” (including modifications such as “assertive program” and “executable program”) to mean “procedure body.” In Section 2.1 we define a program (i.e., a procedure body) to be a sequence of statements. The operational statements—procedure call, selection, and iteration—are likely familiar to the reader; they form the core of executable programs. The statements that permit programs to be assertive may be less familiar. Care will be taken to describe the statements that are important in the intermediate forms of the indexed method because these statements are new with this work. Section 2.1 gives an informal semantics for the statements along with their formal syntax. The formal semantics are presented in Section 2.2.

## 2.1 Syntax

### 2.1.1 Aspects of the Syntax That Are Context-Free

The context-free aspects of the syntax are defined here by a grammar in extended Backus-Naur form (EBNF), the extensions being the use of square brackets (“[” and “]”) as metasympols, indicating that the enclosed sequence of symbols is optional, and

Table 1: Nonterminal Symbols Whose Definitions Are Assumed

<i>symbol</i>	<i>description</i>	<i>example</i>
$\langle p\_nm \rangle$	procedure name	Inv_Abs
$\langle cur\_var \rangle$	current variable name	catalyst
$\langle old\_var \rangle$	old variable name	#catalyst
$\langle ind\_var \rangle$	indexed variable name	catalyst <sub>5</sub>
$\langle cur\_var\_list \rangle$	list of current variable names	q1, catalyst
$\langle b\_p\_e \rangle$	Boolean-valued program expression	q1_empty <b>and</b> q2_empty
$\langle cur\_assert \rangle$	assertion, current variables	x = 7
$\langle old\_assert \rangle$	assertion, current and old variables	#x = 7 $\wedge$ x = 49
$\langle idx\_assert \rangle$	assertion, indexed variables	x <sub>0</sub> = 7 $\wedge$ x <sub>3</sub> = 49
$\langle nat\_num \rangle$	natural number in decimal notation	9856

the use of braces (“{” and “}”) as metasympols, indicating that the enclosed sequence of symbols occurs any number of times (zero or more) in succession [39, pages 21–22]. The syntax we propose is generic with respect to appropriate definitions (e.g., grammars) for the nonterminal symbols shown in Table 1, with the restriction that an old variable name is a current variable name preceded by exactly one old sign (#):

$$\langle old\_var \rangle ::= \# \langle cur\_var \rangle \quad (2.1)$$

The old sign (#) was introduced at the beginning of Section 1.5.3 on page 24.

A program (i.e., a procedure body) is a sequence of zero or more statements:

$$\langle program \rangle ::= \{ \langle stmt \rangle \} \quad (2.2)$$

There are nine different statements; we display their nonterminal symbols here in the rewrite rule for  $\langle stmt \rangle$ , and describe each one in the discussion that follows.

$$\begin{aligned} \langle stmt \rangle ::= & \langle call \rangle \mid \langle selec \rangle \mid \langle loop \rangle \\ & \mid \langle confirm \rangle \mid \langle assume \rangle \mid \langle remember \rangle \\ & \mid \langle whenever \rangle \mid \langle stow \rangle \mid \langle alter\_all \rangle \end{aligned} \quad (2.3)$$

### Operational Statements

We use the usual syntax for procedure call:

$$\langle call \rangle ::= \langle p\_nm \rangle (\langle cur\_var\_list \rangle) \quad (2.4)$$

Calling a procedure has the effect of changing the values of the actual parameters in  $\langle \text{cur\_var\_list} \rangle$ , and any other referenced state variables, according to the definition provided by the procedure's body. This is the usual meaning accorded to a procedure call. We will augment this meaning when we discuss handling the assertive part of assertive programs (see p. 36).

The selection ( $\langle \text{selec} \rangle$ ) and iteration ( $\langle \text{iter} \rangle$ ) statements are compound statements in that they each contain statement sequences. The selection statement has the usual syntax and meaning.

$$\begin{aligned} \langle \text{selec} \rangle ::= & \text{ if } \langle \text{b\_p\_e} \rangle \text{ then} & (2.5) \\ & \{ \langle \text{stmt} \rangle \} \\ & [\text{ else} \\ & \{ \langle \text{stmt} \rangle \}] \\ & \text{ end if} \end{aligned}$$

The Boolean-valued program expression ( $\langle \text{b\_p\_e} \rangle$ ) is first evaluated. If the result is true, the statement sequence in the **then** part is executed, followed by the statements after the **end if**. If the result is false, the statement sequence in the **else** part (if present) is executed, followed by the statements after the **end if**.

The syntax of the iteration statement is a variation of the **loop** statement of Ada.

$$\begin{aligned} \langle \text{iter} \rangle ::= & \text{ loop} & (2.6) \\ & \text{ maintaining } \langle \text{old\_assert} \rangle \\ & \text{ while } \langle \text{b\_p\_e} \rangle \text{ do} \\ & \{ \langle \text{stmt} \rangle \} \\ & \text{ end loop} \end{aligned}$$

A syntactic slot introduced by the keyword **maintaining** provides a place for the loop invariant assertion. This assertion plays a role in the assertive part of assertive programs (see p. 36). Otherwise, the meaning of the iteration statement is the same as for Pascal's **while** statement. An execution begins with evaluation of the Boolean-valued program expression ( $\langle \text{b\_p\_e} \rangle$ ). If the result is true, the statement sequence of the body of the iteration statement is executed, followed by another execution of the iteration statement. If the result is false, execution moves on to the statements after the **end loop**.

The three kinds of operational statements ( $\langle \text{call} \rangle$ ,  $\langle \text{selec} \rangle$ , and  $\langle \text{loop} \rangle$ ) are the only statements that a programmer may write that have the effect of changing the values associated with the current variables during execution. A programmer may write a

**confirm** statement, but execution of a **confirm** statement cannot change the value of a current variable. Execution of a **confirm** statement is a matter for the assertive part of assertive programs—a promised topic to which we now turn.

### Statements That Permit Programs To Be Assertive

In the course of program execution, the assertion in a **confirm** statement may evaluate as false. Such an evaluation may be a witness to the program’s invalidity. That is to say, subject to certain assumptions, the assertive program pledges that the assertion in a **confirm** statement will never evaluate as false. If all the assumptions have held true, a false **confirm** statement violates the pledge, rendering the program invalid.

We must have a way to record and explain an assertive program’s pledges. The usual method of recording and explaining a program’s behavior uses a function from the set of variable names into the universe of values; this function is called the program’s *state*. The effect of a program statement is given by explaining how the statement changes the program’s state. We use an expanded notion called the program’s *environment* to explain the meaning of assertive programs. The environment comprises several components, including the *state* of the current variables. It also includes an *assert-status* to record and explain the pledges an assertive program makes [38, p. 67][12, p. 158][9, p. 8].

When the execution of an assertive program has not yet violated any assumptions, and a **confirm** statement evaluates as false, this fact is recorded by setting the assert-status to the value *categorically false*; the assertive program’s pledge has been violated. The remainder of the program’s execution cannot atone for this violation, so the assert-status must remain at categorically false. That is to say, categorically false is a “stuck state” for the assert-status.

An assertive program states its assumptions in **assume** statements. When the execution of an assertive program has not yet set the assert-status to categorically false, and an **assume** statement evaluates as false, one of the program’s assumptions has been violated. In this case, the assertive program is relieved of any further obligations. This situation is recorded by setting the assert-status to the stuck-state value of *vacuously true*.

An assert-status value is needed for indicating that neither assumption nor pledge has been violated; call this value *neutral*. Execution of a program statement never changes a non-neutral assert-status. Execution of an **assume** statement that evaluates as false changes a neutral assert-status to vacuously true. Execution of a **confirm** statement that evaluates as false changes a neutral assert-status to categorically false.

When the assertion in either of these statements evaluates as true, the assert-status is left unchanged.

A programmer may write a **confirm** statement having an assertion that deals only with current variables. This is permitted as an aid to documentation. Also, some such assertions may be checkable at execution time. A programmer may use neither old nor indexed variables in a **confirm** statement. Programmers never write indexed variables; they arise only during application of the indexed method. The indexed method permits programmers to use old variables only in the postconditions of procedure specifications and in the loop invariants of the **maintaining** clauses.

Application of Krone's proof rules produces **assume** and **confirm** statements whose assertions may contain old and current variables, while application of the indexed method produces **assume** and **confirm** statements whose assertions may contain indexed variables only. Hence, the general syntax of these two statements permits them to contain any of the three kinds of assertions.

$$\langle \text{confirm} \rangle ::= \mathbf{confirm} \langle \text{assert} \rangle \quad (2.7)$$

$$\langle \text{assume} \rangle ::= \mathbf{assume} \langle \text{assert} \rangle \quad (2.8)$$

$$\langle \text{assert} \rangle ::= \langle \text{cur\_assert} \rangle \mid \langle \text{old\_assert} \rangle \mid \langle \text{idx\_assert} \rangle \quad (2.9)$$

The environment we use to explain the meaning of assertive programs includes, of course, the state of indexed variables and the state of old variables.

Because they actively affect the execution of assertive programs by changing the assert status, we chose verbs as the names of the **assume** and **confirm** statements. These two statements correspond, respectively, to the **fact** and **oblig** keywords used in Chapter I. We chose nouns for these keywords because they marked statements of facts and obligations, which are objects.

We are now in a position to explain the effect of a procedure call in connection with the assert-status. In an environment having a non-neutral assert-status, the effect of any statement, including a procedure call, is to leave the environment unchanged. In the case of a neutral assert-status, if the procedure's precondition is violated by the values of the current variables in the environment, the assert-status is set to categorically false and the remainder of the environment is left unchanged. Otherwise, the new values of the current variables, and the new value of the assert-status, are determined by the semantics of that procedure's declaration.

The effect of the iteration statement on a neutral assert-status is as follows. An execution begins with evaluation of the the assertion in the **maintaining** clause (i.e., the intended loop invariant). If the result is true, the Boolean-valued program expression in the **while** clause ( $\langle \text{b\_p\_e} \rangle$ ) is evaluated. If it evaluates to false, execution moves on to the statements after the **end loop**. Otherwise, the statement sequence

of the body of the iteration statement is executed, followed by another execution of the iteration statement. If the result of evaluating the assertion in the **maintaining** clause is false, the assert-status is set to categorically false, and execution moves on to the statements after the **end loop**.

Old variable names (e.g.,  $\#x$ ) are used in recording the state of current variables for reference later in the program's execution. An important use of old variable names is recording the values of formal parameters just prior to execution of the procedure body. This is done so that the truth of the procedure's postcondition, which typically asserts a relationship between parameters' initial and final values, can be evaluated. The **remember** statement records the current state of each current variable in its corresponding old variable name. For example, if the current values of  $x$  and  $y$  are 2 and 8, respectively, then execution of the **remember** statement sets the value of  $\#x$  to 2 and the value of  $\#y$  to 8. The syntax of the **remember** statement is simply the one keyword:

$$\langle \text{remember} \rangle ::= \mathbf{remember} \quad (2.10)$$

Programmers may not write **remember** statements. They arise by application of the proof rules of Krone's system.

### Statements Needed for the Indexed Method's Intermediate Forms

The remaining three statements ( $\langle \text{whenever} \rangle$ ,  $\langle \text{stow} \rangle$ , and  $\langle \text{alter\_all} \rangle$ ) are not written by programmers, but arise by application of the indexed method's proof rules. They disappear again by the time the final mathematical assertion is produced. The **whenever** statement is a compound statement; it contains a statement sequence. It differs from the selection statement in two ways. Its test expression is a mathematical expression, not a program expression. It does not have an optional **else** part. All the variables in the test expression ( $\langle \text{idx\_assert} \rangle$ ) are indexed variables.

$$\begin{aligned} \langle \text{whenever} \rangle ::= & \mathbf{whenever} \langle \text{idx\_assert} \rangle \mathbf{do} & (2.11) \\ & \{ \langle \text{stmt} \rangle \} \\ & \mathbf{end\ whenever} \end{aligned}$$

The meaning of a **whenever** statement is similar to that of an **if-then** statement. The test expression ( $\langle \text{idx\_assert} \rangle$ ) is first evaluated. If the result is true, the statement sequence between **do** and **end whenever** is executed, followed by the statements after the **end whenever**. If the result is false, the **whenever** statement has no effect (except to continue execution with the statements after the **end whenever**).

The indexed method's proof rules use the **whenever** statement to construct branch conditions, a concept discussed in Chapter I.

Essential to the indexed method is the ability to refer to the values current variables had the last time execution reached any arbitrary point in the program. That is why the method associates each place between consecutive programmer-written statements with an index. This is accomplished with the **stow** statement. The effect of the **stow**( $i$ ) statement is to copy the value of each current variable to the corresponding variable with index  $i$ . For example, if the current values of  $x$  and  $y$  are 2 and 8, respectively, then execution of the **stow**(5) statement sets the value of  $x_5$  to 2 and the value of  $y_5$  to 8. The syntax of the **stow** statement is:

$$\langle \text{stow} \rangle ::= \text{stow}(\langle \text{nat\_num} \rangle) \quad (2.12)$$

The indexed method transforms programs to mathematical assertions. Doing so means removing operational statements—statements that change the values of current variables. In the intermediate stages, the removed operational statements must be replaced with a statement that permits the values of the current variables to change; the current variables cannot be left frozen at their previous values. The **alter all** statement does what is needed here. The effect of executing the **alter all** statement can be thought of as giving each current variable some arbitrary value of the right type. The syntax of the **alter all** statement is simply the pair of keywords:

$$\langle \text{alter\_all} \rangle ::= \text{alter all} \quad (2.13)$$

The context-free aspects of assertive-program syntax are collected in Figure 24.

### 2.1.2 Aspects of the Syntax That Are Not Context-Free

We adopt the usual non-context-free syntactic restrictions for programs; e.g., each actual parameter must have the same type as its corresponding formal parameter. To be syntactically correct, programs must also obey the additional restriction that, in any given procedure call, an identifier may appear at most once in the list of variable names; duplicate actual parameters are not permitted. The procedure's *referenced state variables* (i.e., “global variables”) are considered actual parameters in determining whether there is such duplication. We illustrate what we mean by this restriction, and the reason for it, using two examples.

Consider the procedure `Set_State_by_Addition` specified in Figure 25. Let us assume for these examples that variables  $a$  and  $b$  have been declared to be of type `Integer`. With the above-stated restriction in force, the following procedure call would

```

⟨program⟩ ::= {⟨stmt⟩}
⟨stmt⟩ ::= ⟨call⟩ | ⟨selec⟩ | ⟨loop⟩
           | ⟨confirm⟩ | ⟨assume⟩ | ⟨remember⟩
           | ⟨whenever⟩ | ⟨stow⟩ | ⟨alter_all⟩
⟨call⟩ ::= ⟨p_nm⟩(⟨cur_var_list⟩)
⟨selec⟩ ::= if ⟨b_p_e⟩ then
           {⟨stmt⟩}
           [else
            {⟨stmt⟩}]
           end if
⟨iter⟩ ::= loop
           maintaining ⟨old_assert⟩
           while ⟨b_p_e⟩ do
           {⟨stmt⟩}
           end loop
⟨confirm⟩ ::= confirm ⟨assert⟩
⟨assume⟩ ::= assume ⟨assert⟩
⟨assert⟩ ::= ⟨cur_assert⟩ | ⟨old_assert⟩ | ⟨idx_assert⟩
⟨remember⟩ ::= remember
⟨whenever⟩ ::= whenever ⟨idx_assert⟩ do
             {⟨stmt⟩}
             end whenever
⟨stow⟩ ::= stow(⟨nat_num⟩)
⟨alter_all⟩ ::= alter all

```

Figure 24: Context-free Grammar of Assertive Programs

```

procedure Set_State_by_Addition (x, y : Integer)
  referenced state variables z : Integer
  requires MIN_INT ≤ x + y ∧ x + y ≤ MAX_INT
  ensures z = x + y ∧ x = #x ∧ y = #y

```

Figure 25: Specification of Procedure Set\_State\_by\_Addition

not be permitted:

$$\text{Set\_State\_by\_Addition (b, z)} \quad (2.14)$$

The restriction applies to this case due to  $z$ , a referenced state variable of Set\_State\_by\_Addition, appearing as an actual parameter. The reason this can be a problem becomes apparent when we substitute the actual parameters into the procedure's postcondition, yielding three equations.

$$z = b + z \quad (2.15)$$

$$b = \#b \quad (2.16)$$

$$z = \#z \quad (2.17)$$

In general,  $b$  can have nonzero values. But this generality is ruled out by equation 2.15, which forces  $b$  to zero. One might object that equation 2.15 would be better expressed using the old sign ( $\#$ ):

$$z = b + \#z \quad (2.18)$$

However, equation 2.17 makes this a distinction without a difference.

The trouble arose because we expressed the postcondition with three seemingly independent variables,  $z$ ,  $x$ , and  $y$ , but our choice of (duplicate) actual parameters introduced a dependency. The value of  $z$  cannot both be preserved (equation 2.17) and increased by  $b$  (unless  $b$  happens to be zero). One might try to deal with this problem by introducing a complex specification system that produces a different specification depending on the choice of actual parameters. For example, such a system might state that the actual parameter corresponding to the formal parameter  $y$  is preserved *unless* it happens to be  $z$ , in which case it is increased by the value of the actual parameter corresponding to  $x$ . We prefer to deal with this problem, as Cook did [4, p. 76], by prohibiting duplicate actual parameters. Doing so assures that the different variables used to express a postcondition will not obtain dependencies from the choice of actual parameters.

```

procedure Add (z, x, y : Integer)
  requires MIN_INT ≤ x + y ∧ x + y ≤ MAX_INT
  ensures z = x + y ∧ x = #x ∧ y = #y

```

Figure 26: Specification of Procedure Add

This restriction can rule a procedure call syntactically incorrect even when the called procedure has no referenced state variables. Neither of these calls to procedure Add of Figure 26 is permitted, even though call 2.20 happens to present no problem for this particular specification. (It might, however, be trouble for some implementations of Add!)

$$\text{Add (a, b, a)} \tag{2.19}$$

$$\text{Add (b, a, a)} \tag{2.20}$$

## 2.2 Semantics

The *semantics* of a programming language defines what a computer is supposed to do when it executes a program written in the language. This section gives a semantics for the assertive programs whose syntax is defined in Section 2.1. Only the operational statements ( $\langle \text{call} \rangle$ ,  $\langle \text{selec} \rangle$ , and  $\langle \text{iter} \rangle$ ) are intended to be executed by real computers, although the choice in some systems might be to also execute a class of **confirm** ( $\langle \text{cur\_assert} \rangle$ ) statements for testing and debugging purposes. The other statements arise temporarily in the translation from procedure body to mathematical assertion when the proof rules of Chapter III are applied. The semantics given here also define how these intermediate forms should be “executed,” this definition being important to our argument for the soundness and relative completeness of the proof rules, which we discuss in Chapter IV.

Earlier in the history of programming languages, some researchers [18, 13] were inclined to define a language’s semantics in terms of proof rules. This kind of semantic definition came to be called *axiomatic semantics* [39, p. 193]. These researchers were attracted to the substantial advantages [18, pp. 579, 580, and 583] of having a deductive system—a calculus—consisting of proof rules for reasoning about program behavior. The direct route to obtaining such a deductive system—with its advantages—was simply to let the proposed proof rules define the language’s semantics.

It was not too long before some of these same researchers noted shortcomings of this direct approach. It is not entirely clear how to establish whether a given compiler and run-time system satisfies a set of proof rules. A 1974 paper by Hoare and Lauer [21, p. 136] suggested

... an approach to the solution of these problems. It is based on the realization that a single formal definition is unlikely to be equally acceptable to both implementor and user, and that at least two definitions are required, a constructive one to act as a guide and model for the implementor, and an implicit one for the user, who is interested in what his program accomplishes, as much as in how it does it. The doubt arises whether the two descriptions describe the same language; but this doubt can be completely resolved by a mathematical proof of the consistency of the two definitions; and then the pair of complementary definitions can serve together as an interface between implementor and user, which is just as rigorous but possibly more acceptable to each of them than a single definition could be.

Furthermore, if a set of proof rules is inconsistent, then it is useless: incorrect assertions can be proved in such a system. In 1978, Cook [4, p. 70] noted:

The rules for procedure call statements often (in fact usually) have technical bugs when stated in the literature, and the rules stated in earlier versions of the present paper are not exceptions. In the process of trying to prove the soundness of these rules, I uncovered some of the bugs, and this led me to believe a careful and detailed proof of soundness is necessary to have any confidence that there are no further bugs.

Cook [4, p. 70] approached his justification of the soundness (i.e., consistency) of his axiom system “by introducing an interpretive semantics for the language.” By associating each program with a well-defined function, his semantics served to define a notion of *truth*, answering the question: “What exactly does it mean to execute a given program?” Cook’s axiom system, on the other hand, served to define the separate notion of *proof*, answering the different question: “Which program-specification pairs can be derived according to the rules of the system?” In the tradition of the twentieth-century development of mathematical logic, he recognized the value of separating these two notions (truth and proof); having been distinguished, they could be compared and contrasted with each other. Cook used comparison to show that his definition of proof is consistent with his definition of truth. Further evidence that Cook separated semantics from proof rules is that, although he included “axiomatic

semantics” in his paper’s list of key words, he did not use the term in the paper’s text, preferring the term “axiom system.”

Contrasts between Cook’s notions of truth and proof help us see the value gained by separating them and showing proof to be consistent with truth. Defining the association of a function with each program is a straightforward enterprise, admitting few surprises; hence, his definition of truth is simple. On the other hand, interactions among axioms and rules of an axiom system are complex—not easily predicted. We can understand and evaluate the notion of truth more easily than that of proof. Another contrast is that manipulations and reasoning are more easily performed in an axiom system than they are using the function definitions of the notion of truth. We can work more effectively using proof than we can using truth. This effectiveness is due to the fact that the axiom system (proof) leaves out unnecessary details involved in the semantics (truth); the axiom system is more abstract than the semantics.

Hence, we can evaluate the notion of truth, effectively determining if its definition is adequate and correct. Then, having shown the more abstract notion of proof to be consistent with the notion of truth, we can work effectively in the proof system, confident that all things we properly prove are also true. These are reasons why Cook established a less abstract semantics for the programming language as a basis for justifying the soundness of the proof rules.

We use sets, functions, and relations to define the semantics of assertive programs because set theory is well understood, having been widely used this century for many purposes, including reasoning about mathematics. That is to say, set theory is commonly used for doing *meta*-mathematics, with great success. In contrast, a new deductive system, such as the one proposed in Chapter III, is not well understood. Without further examination, we do not know whether such a deductive system is sound. Our strategy is to define the semantics of assertive programs on the firm foundation of set theory, and, then, in Chapter IV, discuss our set-theoretic proof that Chapter III’s deductive system is sound. Having shown its soundness, we can, henceforth, confidently use the convenience of the deductive system to reason about assertive programs.

### 2.2.1 Semantic Space

Programs usually have been interpreted as mappings from states into states, where a state gives the current values of the program variables. Such an interpretation is called a *denotational semantics* [39, pp. 167–93]. Navlakha [38, p. 67] and Ernst et al. [12, p. 158][9, p. 8] have shown both the need for and the utility of a richer semantic space for describing the effect of executing programs containing declarations of the

specifications of external procedures. We shall see in Chapter IV that their idea of *assert-status* is a crucial tool for showing the soundness and relative completeness of the indexed method. The semantic space we need here is an augmented form of the space Ernst et al. used in [10, pp. 267–270], and its various domains are given in Figure 27. The reader is likely to be unfamiliar with this richer semantic space, so we provide the following explanations associated with various parts of Figure 27.

### Figure 27, Part 1: Interpretation $\mathcal{I}$

We interpret programs as mappings from environments to environments. Assertions (the contents of **maintaining** clauses and **assume** and **confirm** statements) and Boolean-valued program expressions ( $\langle b\_p\_e \rangle$ ) are interpreted by function  $\mathcal{I}$  as mappings from environments to true or false (Boolean) because they have no effect on the environment.

### Figure 27, Part 2: Environment

An environment consists of six kinds of information about the execution of a program. The most obvious part of an environment is the current-state, which maps the names of program variables (i.e., current variables) into their values (part 4 of Figure 27). Because the current-state is part of the environment, and we are interpreting programs as mappings from environments to environments, the kind of semantics we are providing is, still, denotational.

### Figure 27, Part 3: Assert-status

The assert-status keeps track of the effect of executing the assertions in a program. Certain assertions in a program must be true in order for it to be correct, e.g., the precondition of a procedure called by the program. When such an assertion is violated, the assert-status becomes categorically false (CF), and remains so for the rest of the program's execution. Thus, the violation of a condition necessary for correctness is indelibly recorded. The **confirm** statement, which arises during application of the proof rules, contains an assertion that must be true.

On the other hand, a program's correctness can rely on certain other assertions being true. For example, if the program includes a specification of a procedure, declaring this procedure to be externally defined, then the program assumes that the environment contains a meaning for that procedure that matches the specification. When such an assumption is violated, the assert-status becomes, and remains, vacuously true (VT). Further execution of the program in an environment that contains

1.  $\mathcal{I} : (\text{Programs}) \rightarrow (\text{Environments} \rightarrow (\text{Environments} \cup \text{Boolean}))$
2.  $\text{Environments} = \text{Assert-statuses} \times \text{Current-states} \times \text{Old-states} \times$   
 $\text{Index-states} \times \text{Setups} \times \text{Declaration-meanings}$
3.  $\text{Assert-statuses} = \{\text{VT}, \text{CF}, \text{NL}\}$
4.  $\text{Current-states} = (\text{Current-variable-names} \rightarrow \text{Values})$
5.  $\text{Index-states} = (\text{Integers} \rightarrow \text{Current-states})$
6.  $\text{Setups} = \text{Current-states}^*$
7.  $\text{Old-states} = (\text{Augmented-old-variable-names} \rightarrow \text{Values})$
8.  $\text{Declaration-meanings} = (\text{Identifiers} \rightarrow (\text{Type-meanings} \cup \text{Predicate-meanings} \cup$   
 $\text{Procedure-meanings} \cup \text{Generic-meanings} \cup \text{Module-meanings}))$
9.  $\text{Values} = \{v \mid \text{there exists } t \in \text{Type-meanings} \text{ such that } v \in t\}$
10.  $\text{Type-meanings} = \text{Base-types} \cup \text{Defined-types}$
11.  $\text{Predicate-meanings} = \{p \mid p : d \rightarrow t \wedge d \in \text{Argument-domains} \wedge$   
 $t \in \text{Type-meanings}\}$
12.  $\text{Argument-domains} = \{T_1 \times \dots \times T_n \mid T_i \in \text{Type-meanings} \wedge 1 \leq n \wedge 1 \leq i \leq n\}$
13.  $\text{Procedure-meanings} = \text{Predicate-meanings} \times \text{Predicate-meanings} \times$   
 $\text{Procedure-functions} \times \text{Status-functions}$
14.  $\text{Procedure-functions} = \{f \mid f : d \rightarrow r \wedge d \in \text{Argument-domains} \wedge$   
 $r \in \text{Argument-domains}\}$
15.  $\text{Status-functions} = \{f \mid f : d \rightarrow \text{Assert-statuses} \wedge d \in \text{Argument-domains}\}$

Figure 27: Definition of Domains in the Semantic Space

a non-matching procedure meaning is of no use in determining the program's correctness. That is why such environments are "filtered out" by setting the assert-status to vacuously true (VT). The **assume** statement, which arises during application of the proof rules, contains an assertion that can be assumed to be true.

The neutral assert-status (NL) indicates that no assertion in a program has been violated so far. We define a *neutral environment* to be one in which assert-status equals NL. A *categorically false environment* has a CF assert-status, and the assert-status of a *vacuously true environment* equals VT.

### Figure 27, Part 5: Index-state

As discussed in Section 1.5.2, the indexed method employs many new variable names, obtained by using the unique integers as subscripts on the names of the program variables. An index-state maps each integer to a current state, which is a mapping from current variable names to values. For example, let  $x$  be an Integer program variable, and let  $x_3$  be one of the indexed variables associated with it. Let  $ns$  be an index-state. Then  $ns(3)$  is a current state. If  $ns(3)$  maps  $x$  to 275 (i.e., if  $ns(3)(x) = 275$ ), the value of  $x_3$  is 275.

### Figure 27, Part 6: Setup

We employ the "setup" to help define the meaning of the **alter all** statement. A setup is a finite sequence of current-states (as indicated by our use of the Kleene \* in the figure). The **alter all** statement changes the current-state and the setup as follows: it makes the new value of the current-state become the state which is at the front of the setup (sequence), and removes this state from the front of the setup. The term "setup" is a slang expression for a situation (e.g., a trial jury!) that has been "rigged" (arranged in advance). The initial environment contains, in the setup component, a prearrangement of the meaning of each **alter all** statement that might be encountered during execution.

### Figure 27, Part 7: Old-state

Depending on whether an old variable,  $\#\xi$ , occurs in an **ensures** clause or a **maintaining** clause, it refers to the value of parameter  $\xi$  at procedure invocation or to the value of variable  $\xi$  at the beginning of the loop. The name of an old variable contains exactly one old sign ( $\#$ ), and it is always the first character (see page 33). Because loops occur inside procedures and can be nested within one another, interpreting these programs requires remembering the values of old variables on a last-in-first-out basis. We can use an expanded name space to accomplish this task. Therefore, as an

aid to defining old-states, we define a new set, “Augmented-old-variable-names,” as the set of all names that can be rewritten from the nonterminal symbol  $\langle \text{aug\_old\_var} \rangle$  in our grammar with the additional rewrite rule:

$$\langle \text{aug\_old\_var} \rangle ::= \{ \# \} \# \langle \text{cur\_var} \rangle \quad (2.21)$$

An augmented old variable name has one or more old signs ( $\#$ ) in its prefix. We illustrate the use of augmented old variable names in our discussion of the semantics of iteration, which begins on page 54.

### Figure 27, Part 8: Declaration-meaning

A declaration meaning maps certain global names into their semantics. These global names include types, procedures, functions, generics, and modules. Depending on the programming language, other kinds of objects may have to be included in the declaration meanings. This dissertation is concerned only with using the meanings of these objects in defining the meaning of procedure bodies. In our semantics for procedure bodies, their executions make no change to the declaration meanings, but are, of course, affected by them.

How program declarations establish the value of declaration-meanings is discussed in other papers [38, 12, 9, 10, 11, 25]. These papers do not attempt to handle modules because the problem of module semantics has not been adequately solved for their purposes. Neither they nor we deny the importance of modules.

### Figure 27, Parts 9 and 10: Value and Type-meaning

The set of values that a variable of a given type can have is the semantics of that type. We call the predefined types “base types.” The types that are used to define abstract types vary from application to application; we call them “defined types.” The collection of all values of all types is the set we call “Values.”

### Figure 27, Parts 11 and 12: Predicate-meaning

The only difference between predicates and functions of the specification language is that possible values that can be produced by applying a function to legal arguments can be of any one type, not just Boolean, as is the case for predicates. Therefore, we combine the semantics of predicates and functions of the specification language in Figure 27, parts 11 and 12. A predicate-meaning whose range is  $\{\mathbf{true}, \mathbf{false}\}$  is what is normally called a predicate; other members of predicate-meanings correspond to functions of the specification language. We defer discussion of parts 13, 14, and 15

AE : Environments  $\rightarrow$  Assert-statuses  
 CSE : Environments  $\rightarrow$  Current-states  
 ISE : Environments  $\rightarrow$  Index-states  
 SPE : Environments  $\rightarrow$  Setups  
 OSE : Environments  $\rightarrow$  Old-states  
 DME : Environments  $\rightarrow$  Declaration-meanings

Figure 28: Six Projection Functions on Environments

of Figure 27 (which help define the meaning of procedure calls) to the next section (2.2.2), where we define our language's semantics.

### 2.2.2 Semantic Definition

We define, in this section, the function  $\mathcal{I}$  of Figure 27, part 1. Doing so establishes the meaning of any syntactically correct program (as defined in Section 2.1). In the following definitions, we let  $\text{env}$  stand for an arbitrary environment, and  $S1$  for a statement ( $\langle \text{stmt} \rangle$ ). Each of  $SS$ ,  $SS1$ , and  $SS2$  stands for a (possibly empty ( $\varepsilon$ )) sequence of statements ( $\{\langle \text{stmt} \rangle\}$ ). We will need to refer to the various components of any environment. The six projection functions shown in Figure 28 allow us to do so. Figure 29 expresses a convenient notation for arbitrary environment  $\text{env}$ , i.e.,

$$\text{env} = [a, cs, ns, se, os, d] \tag{2.22}$$

In the usual fashion, we define the interpretation of a sequence of statements as the interpretation of the tail sequence in the environment resulting from interpreting the first statement in the sequence:

$$\mathcal{I}(S1 \ SS2)(\text{env}) \stackrel{\text{def}}{=} \mathcal{I}(SS2)(\mathcal{I}(S1)(\text{env})) \tag{2.23}$$

The interpretation of the empty sequence of statements is the identity function:

$$\mathcal{I}(\varepsilon)(\text{env}) \stackrel{\text{def}}{=} \text{env} \tag{2.24}$$

The interpretation of any statement in either a vacuously true or a categorically false environment is the identity function; the environment remains unchanged by the

AE(env)	=	a	(Assert-status)
CSE(env)	=	cs	(Current-state)
ISE(env)	=	ns	(Index-state)
SPE(env)	=	se	(Setup)
OSE(env)	=	os	(Old-state)
DME(env)	=	d	(Declaration-meaning)

Figure 29: Notation for Environment Named “env”

interpretation:

$$\mathcal{I}(S1)([VT, cs, ns, se, os, d]) \stackrel{\text{def}}{=} [VT, cs, ns, se, os, d] \quad (2.25)$$

$$\mathcal{I}(S1)([CF, cs, ns, se, os, d]) \stackrel{\text{def}}{=} [CF, cs, ns, se, os, d] \quad (2.26)$$

We now need only define the interpretation of each nonempty statement in a neutral environment. So, in the remaining definitions, let

$$\text{env}_{\text{NL}} = [\text{NL}, cs, ns, se, os, d] \quad (2.27)$$

The only effect of the **stow**( $i$ ) statement is to change the value of the index-state at  $i$  to equal the current-state  $cs$ :

$$\mathcal{I}(\text{stow}(i))(\text{env}_{\text{NL}}) \stackrel{\text{def}}{=} [a, cs, ns', se, os, d] \quad \text{where} \quad (2.28)$$

$$ns'(k) = \begin{cases} ns(k) & \text{if } k \neq i \\ cs & \text{if } k = i \end{cases} \quad (2.29)$$

The main purpose of the **alter all** statement is to change the value of the current-state; it changes the setup so that the next **alter all** statement executed can change the current-state to yet another state. An **alter all** statement will change a neutral environment whose setup is empty to a vacuously true environment because such an environment does not provide sufficient information to further evaluate the validity of the program. In other words, a characteristic of truly useful initial environments is that their setups are long enough to handle all the **alter all** statements encountered during execution:

$$\mathcal{I}(\text{alter all})(\text{env}_{\text{NL}}) \stackrel{\text{def}}{=} [a', cs', ns, se', os, d] \quad \text{where} \quad (2.30)$$

$$a' = \begin{cases} VT & \text{if } se = \varepsilon \\ NL & \text{if } se \neq \varepsilon \end{cases} \quad (2.31)$$

$$cs' = \begin{cases} cs & \text{if } se = \varepsilon \\ \text{first}(se) & \text{if } se \neq \varepsilon \end{cases} \quad (2.32)$$

$$se' = \begin{cases} se & \text{if } se = \varepsilon \\ \text{tail}(se) & \text{if } se \neq \varepsilon \end{cases} \quad (2.33)$$

Let  $Q$  be an assertion ( $\langle \text{cur\_assert} \rangle$ ,  $\langle \text{old\_assert} \rangle$ , or  $\langle \text{idx\_assert} \rangle$ ). We define  $\mathcal{I}(Q)(\text{env})$  to be the usual interpretation of  $Q$  as a predicate logic expression, where the assignments to the free variables are determined by  $cs$ ,  $ns$ , and  $os$ . Consequently,  $\mathcal{I}(Q)(\text{env}) \in \{\text{true}, \text{false}\}$ . Interpreting **assume**  $Q$  and **confirm**  $Q$  in a neutral environment, say  $\text{env}_{\text{NL}}$ , affects only the assert-status, which is changed if and only if  $\mathcal{I}(Q)(\text{env}_{\text{NL}})$  is false:

$$\mathcal{I}(\mathbf{assume} \ Q)(\text{env}_{\text{NL}}) \stackrel{\text{def}}{=} [a', cs, ns, se, os, d] \quad \text{where} \quad (2.34)$$

$$a' = \begin{cases} \text{NL} & \text{if } \mathcal{I}(Q)(\text{env}_{\text{NL}}) \\ \text{VT} & \text{if } \neg \mathcal{I}(Q)(\text{env}_{\text{NL}}) \end{cases} \quad (2.35)$$

$$\mathcal{I}(\mathbf{confirm} \ Q)(\text{env}_{\text{NL}}) \stackrel{\text{def}}{=} [a', cs, ns, se, os, d] \quad \text{where} \quad (2.36)$$

$$a' = \begin{cases} \text{NL} & \text{if } \mathcal{I}(Q)(\text{env}_{\text{NL}}) \\ \text{CF} & \text{if } \neg \mathcal{I}(Q)(\text{env}_{\text{NL}}) \end{cases} \quad (2.37)$$

Interpreting a **whenever** statement in a neutral environment, depending on the interpretation of the assertion  $Q$  (all of whose variables, according to the syntax, are indexed variables), either leaves the environment unchanged or changes it just as the body of the **whenever** statement would:

$$\begin{aligned} & \mathcal{I}(\mathbf{whenever} \ Q \ \mathbf{do} \ SS \ \mathbf{end} \ \mathbf{whenever})(\text{env}_{\text{NL}}) && (2.38) \\ & \stackrel{\text{def}}{=} \begin{cases} \mathcal{I}(SS)(\text{env}_{\text{NL}}) & \text{if } \mathcal{I}(Q)(\text{env}_{\text{NL}}) \\ \text{env}_{\text{NL}} & \text{if } \neg \mathcal{I}(Q)(\text{env}_{\text{NL}}) \end{cases} \end{aligned}$$

Consider now the interpretation of Boolean-valued program expressions ( $\langle \text{b-p-e} \rangle$ ). Our simple, example programming language includes only *proper* procedures, not *function* procedures, i.e., procedure calls appear only as statements, not in expressions as function calls. Consequently, our Boolean-valued program expressions consist only of program variables, Boolean-operator key words, and parentheses for overriding operator precedence. Therefore, it is easy to define their interpretation to involve no change to the environment. We define  $\mathcal{I}(b\_p\_e)(\text{env})$  to be the usual interpretation of a propositional logic expression, where the assignments to the free variables are determined by  $cs$ . Consequently,  $\mathcal{I}(b\_p\_e)(\text{env}) \in \{\text{true}, \text{false}\}$ .<sup>3</sup> Interpreting an **if-then** statement in a neutral environment, depending on the interpretation of the

<sup>3</sup>The possibility of simplifying the example language by excluding function procedures was suggested by Hollingsworth's dissertation [22] where he established forty-one principles for building

Boolean-valued program expression  $b_{p-e}$ , either leaves the environment unchanged or changes it just as the body of the **if** statement would:

$$\mathcal{I}(\mathbf{if } b_{p-e} \mathbf{ then SS end if})(\text{env}_{\text{NL}}) \stackrel{\text{def}}{=} \begin{cases} \mathcal{I}(\text{SS})(\text{env}_{\text{NL}}) & \text{if } \mathcal{I}(b_{p-e})(\text{env}_{\text{NL}}) \\ \text{env}_{\text{NL}} & \text{if } \neg \mathcal{I}(b_{p-e})(\text{env}_{\text{NL}}) \end{cases} \quad (2.39)$$

Interpreting an **if-then-else** statement in a neutral environment changes it either just as the body of the **then** portion of the statement or just as the body of the **else** portion of the statement would:

$$\begin{aligned} \mathcal{I}(\mathbf{if } b_{p-e} \mathbf{ then SS1 else SS2 end if})(\text{env}_{\text{NL}}) & \quad (2.40) \\ \stackrel{\text{def}}{=} \begin{cases} \mathcal{I}(\text{SS1})(\text{env}_{\text{NL}}) & \text{if } \mathcal{I}(b_{p-e})(\text{env}_{\text{NL}}) \\ \mathcal{I}(\text{SS2})(\text{env}_{\text{NL}}) & \text{if } \neg \mathcal{I}(b_{p-e})(\text{env}_{\text{NL}}) \end{cases} \end{aligned}$$

### Semantics of Procedure Call

This section follows the explanation given by Ernst et al. [10, pp. 268–269].

The semantics of a procedure has four different components ([part 13, Figure 27]). The first is the domain-predicate which is a predicate on the parameters of the procedure. A procedure is usually designed only for a subset of all possible values for its parameters; the domain-predicate specifies this subset. The second component of a procedure-meaning is the effect-predicate which is an “I/O relation” for the procedure. It is a predicate whose arguments are the values of the input and output parameters of the procedure. The purpose of this predicate is to indicate which values of the output parameters are correct for the input parameter values.

Each procedure also has a procedure-function, [definition 14 in Figure 27]. This function maps the values of input parameters into the values for the output parameters. Finally, the status-function maps the input parameters into the new assert-status for the environment produced by executing the procedure ([definition 15, Figure 27]).

---

high-quality software in Ada. Principle number nine demands that all operations be proper procedures. Hollingsworth’s discipline would not ban function procedures in all languages; it does not ban them in C++, and the research language, RESOLVE, has function procedures. But there are technical problems with function procedures in Ada [22, pp. 52–56]. In RESOLVE, function procedures are specified as preserving the environment; so, any program expression (including those that are Boolean-valued), when evaluated in a neutral environment, leaves the environment unchanged and returns a value.

A procedure-meaning contains the semantics of both the specification and implementation of a procedure; i.e., the domain- and effect-predicates are the semantics of its specification and the semantics of its implementation are the procedure- and status-functions. The semantics of a correct procedure is a procedure-meaning in which the implementation semantics does not violate the semantics of the specification. Such a procedure-meaning is defined to be *conformal* if the following conditions are true: let  $pd = [dp, ep, pf, sf]$  be the procedure-meaning. Then,

1. the number and types of arguments of the various components of  $pd$  are consistent with one another;
2. for all  $x_1, \dots, x_n$  such that  $dp(x_1, \dots, x_n)$  and for all  $y_1, \dots, y_m$  such that  $pf(x_1, \dots, x_n) = [y_1, \dots, y_m]$ ,  
 $sf(x_1, \dots, x_n) \neq CF$  and  
 if  $sf(x_1, \dots, x_n) = NL$  then  $ep(x_1, \dots, x_n, y_1, \dots, y_m)$ .

In these conditions the  $x$ s represent inputs to the procedure and the  $y$ s represent the outputs of the procedure. The first condition requires that the components of a procedure-meaning have arguments which are consistent with one another. For example, the status-function and the procedure-function must have the same number and types of arguments because their arguments are just the input parameters of the procedure. The  $x$ s and  $y$ s in the above conditions show the number of arguments to the various components of  $pd$  and also when one argument must have the same type as another argument. Condition (2) applies to those input parameters  $x_i$  which make the domain-predicate true: the status-function must not be  $CF$  and if it is  $NL$ , then the effect-predicate must also be true of the output parameters  $y_i$  produced by  $pf$ . Intuitively this condition requires the outputs of the procedure to satisfy its postcondition whenever its inputs satisfy its precondition. If the procedure does not terminate, then it has no outputs and the condition is true by default. The condition also states that there is no violation of required assertions such as the precondition of an externally defined procedure when it is invoked inside the procedure. The semantics of such externally defined procedures are given by the declaration-meanings component of the environment.

A procedure declaration essentially defines the various components of a procedure-meaning. The interpretation of a procedure's precondition is, roughly speaking, the domain-predicate of the procedure; a more precise description is given in Ernst et al. [9]. Similarly the interpretation of a

procedure's postcondition gives its effect-predicate. By use of the minimum fixed-point operation, the body of a procedure defines its procedure-function and its status-function in a manner similar to that in Scott and Strachey [43].

The interpretation of a procedure declaration  $D$  in a neutral environment  $\text{env} = [\text{NL}, \text{cs}, \text{ns}, \text{se}, \text{os}, \text{d}]$  is defined as follows:  $\mathcal{I}(D)(\text{env}) = [a', \text{cs}, \text{ns}, \text{se}, \text{os}, \text{d}]$  because the interpretation of  $D$  affects only the assert-status component of the environment [10, p. 269].

The reason it does not affect  $\text{d}$  is that any  $\text{env}$  with the wrong meaning for  $D$  is filtered out by making  $a' = \text{VT}$ . Thus, when  $a'$  is not  $\text{VT}$ , then  $\text{env}$  has the correct meaning for  $D$ . If the procedure-meaning  $\text{pm}$  defined by  $D$  as described above is not conformal, then  $a' = \text{CF}$  because  $D$  is incorrect. If  $\text{pm}$  differs from the procedure-meaning  $\text{d}(p)$  stored in the environment, then  $a' = \text{VT}$ . In the remaining case  $a' = \text{NL}$  because  $D$  is correct and  $\text{env}$  has the intended interpretation of  $D$ . This defines  $\mathcal{I}(D)$  for all environments because once the assert-status becomes  $\text{VT}$  or  $\text{CF}$ , the environment remains unchanged; i.e.,  $\mathcal{I}(D)(\text{env}) = \text{env}$  whenever  $\text{env}$  is not a neutral environment.

This definition applies to all environments, even though some environments are not intended for the declaration. For example,  $D$  may be a procedure with three arguments and  $\text{d}(p)$  may be a procedure with a single argument. The interpretation “filters out” such meaningless environments by setting the assert-status to  $\text{VT}$ . The correctness of a program is determined by its interpretation in meaningful environments and the ones that are filtered out are essentially ignored. [10, p. 269]

We are now in position to define the interpretation of a procedure call. We do so for a procedure that has two formal parameters and one referenced state variable. It is straightforward to use this definition as a guide to the semantics of calling a procedure that has a different number of parameters and referenced state variables. We assume that declaration-meaning  $\text{d}$  of environment  $\text{env}$  contains  $\text{d}(\text{P\_nm}) = [\text{dp}, \text{ep}, \text{pf}, \text{sf}]$ , the procedure-meaning corresponding to the following specification:

$$\begin{aligned}
 &\mathbf{procedure} \text{ P\_nm}(x, y : T_1) && (2.41) \\
 &\quad \mathbf{referenced\ state\ variables} \ z : T_3 \\
 &\quad \mathbf{requires} \ \text{pre}[x, y, z] \\
 &\quad \mathbf{ensures} \ \text{post}[x, \#x, y, \#y, z, \#z]
 \end{aligned}$$

Because we are only defining the interpretation of a call to  $P\_nm$  in a neutral environment (i.e.,  $AE(env) = NL$ ), we have that  $dp$  is the same as  $pre$ ,  $ep$  is the same as  $post$ , and  $d(P\_nm)$  is conformal. Let  $\xi$ -proj be the projection function for the  $\xi$ -component of the range of the procedure function  $pf$ , where  $\xi \in \{x, y, z\}$ . Letting  $ac$  and  $ad$  be two distinct variables of type  $T_1$ , we define:

$$\mathcal{I}(P\_nm(ac, ad))(env) \stackrel{\text{def}}{=} [a', cs', ns, se, os, d] \quad \text{where} \quad (2.42)$$

$$a' = \begin{cases} sf(cs(ac), cs(ad), cs(z)) & \text{if } dp(cs(ac), cs(ad), cs(z)) \\ CF & \text{otherwise} \end{cases}, \text{ and} \quad (2.43)$$

$$cs'(\xi) = \begin{cases} cs(\xi) & \text{if } \neg dp(cs(ac), cs(ad), cs(z)) \\ x\text{-proj}(pf(cs(ac), cs(ad), cs(z))) & \text{else if } \xi = ac \\ y\text{-proj}(pf(cs(ac), cs(ad), cs(z))) & \text{else if } \xi = ad \\ z\text{-proj}(pf(cs(ac), cs(ad), cs(z))) & \text{else if } \xi = z \\ cs(\xi) & \text{otherwise} \end{cases} \quad (2.44)$$

### Semantics of Iteration

In the tradition of denotational semantics [35], we use the minimum fixed point (MFP) of a functional along the way to defining the interpretation of the **while** loop. However, we go further and define the interpretation of an *assertive while* loop. Our syntax requires the inclusion of a loop invariant in the **maintaining** clause. We interpret the loop as asserting the truth of the invariant at the start of each iteration. To be useful, the invariant assertion must be able to refer to “old” values of current variables, i.e., values at the start of the loop’s execution, or earlier. To keep reasoning about a loop local to that loop, we define “old” values of current variables to be just the values at the start of the loop’s execution.

We use the old-state (part 7 of Figure 27) to remember the old values of current variables. Because loops can be nested within one another, at the conclusion of a loop’s execution, we must restore the old-state to its value just prior to the loop’s execution. We call this restoring operation “forgetting.” If  $\xi$  is a current variable, its old value is stored in the old-state as the value of the variable  $\#\xi$  ( $\xi$  preceded by one old sign). The old value for the loop containing the currently executing loop would be stored in the old-state as the value of the variable  $\#\#\xi$  ( $\xi$  preceded by two old signs). We define the old-state to behave as a last-in-first-out stack, using the mechanism of the number of old signs in a variable name’s prefix to express the definition. Recall that variable names having one or more old signs in their prefix are augmented old variable names (page 47).

Figure 30 shows an example of a loop nested inside another. We use this example to illustrate the meaning and use of old variable names, and how augmented old variable names are used in remembering and forgetting old values. The purpose of this assertive program is to set  $r$  to the product of  $m$  and  $n$  without changing  $m$  or  $n$ , assuming both  $m$  and  $n$  are nonnegative. The **remember** statement effectively copies the value of  $m$  into the old variable  $\#m$ , but not before it copies the value of  $\#m$  into the augmented old variable  $\#\#m$ . In this way, the former value of  $\#m$  is not lost. The **remember** statement even remembers the former values of augmented old variables; i.e., it copies the value of  $\#\#m$  into  $\#\#\#m$ , and so on. The **remember** statement performs this service for all variables, including  $n$ ,  $r$ ,  $i$ , and  $j$ .

Execution of a loop also begins with this remembering service. Hence, whenever execution reaches index 9, for example, the following three statements are true: the value of  $\#i$  equals the value  $i$  held the last time execution reached index 6; the value of  $\#\#i$  equals the value  $i$  held the last time execution reached index 3; and  $\#\#\#i$  equals the value  $i$  held the last time execution reached index 0. This behavior is important because we intend the phrase “ $i = \#i$ ” of the second loop invariant to mean that  $i$  remains unchanged from index 6 to index 10, not that  $i$  has the same value it had at index 3, i.e., zero!

Execution of a loop concludes with the opposite of the remembering service: forgetting. Forgetting restores the values of the augmented old variables to the values they had when the loop was encountered. Forgetting does not change the values of current variables! Forgetting sets the value of  $\#m$  to the value of  $\#\#m$ , the value of  $\#\#m$  to that of  $\#\#\#m$ , and so on. This behavior is important because we intend the phrase “ $m = \#m$ ” of the final **confirm** statement to mean that the value of  $m$  at index 11 is the same value it had at index 0.

The meaning of a **while** loop in an environment,  $\text{env}_{\text{NL}}$ , is the composition of three functions: first  $\text{env}_{\text{NL}}$  is remembered, then the minimum fixed point of a functional is applied, and, finally, the “forget” function is applied. Now let us formally define the remember ( $\text{Rem} : \text{Environments} \rightarrow \text{Environments}$ ) and forget ( $\text{Fgt} : \text{Environments} \rightarrow \text{Environments}$ ) functions. Let  $\psi$  stand for an augmented old variable name; let  $\xi$  stand for a current variable name; and let  $k$  be a natural number. Let  $\#^j$  represent  $j$  occurrences of  $\#$ ; e.g.,  $\#^3$  represents  $\#\#\#$ .

$$\text{Rem}(\text{env}_{\text{NL}}) \stackrel{\text{def}}{=} [\text{a}, \text{cs}, \text{ns}, \text{se}, \text{os}', \text{d}] \quad \text{where} \quad (2.45)$$

$$\text{os}'(\psi) = \begin{cases} \text{cs}(\xi) & \text{if } \psi = \#\xi \\ \text{os}(\#\#^{k+1}\xi) & \text{if } \psi = \#\#^{k+2}\xi \end{cases} \quad (2.46)$$

$$\text{Fgt}(\text{env}_{\text{NL}}) \stackrel{\text{def}}{=} [\text{a}, \text{cs}, \text{ns}, \text{se}, \text{os}', \text{d}] \quad \text{where} \quad (2.47)$$

$$\text{os}'(\psi) = \text{os}(\#\psi) \quad (2.48)$$

```

remember
-- (0)
  assume  $0 \leq m \wedge 0 \leq n$ 
-- (1)
   $r := 0$ 
-- (2)
   $i := 0$ 
-- (3)
  loop
    maintaining  $i \leq m \wedge r = i \cdot n \wedge m = \#m \wedge n = \#n$ 
    while  $i < m$  do
-- (4)
       $i := i + 1$ 
-- (5)
       $j := 0$ 
-- (6)
      loop
        maintaining  $j \leq n \wedge r = (i - 1) \cdot n + j \wedge i = \#i \wedge m = \#m$ 
           $\wedge n = \#n$ 
        while  $j < n$  do
-- (7)
           $j := j + 1$ 
-- (8)
           $r := r + 1$ 
-- (9)
        end loop
-- (10)
      end loop
-- (11)
    confirm  $r = m \cdot n \wedge m = \#m \wedge n = \#n$ 

```

Figure 30: Nested Loops and Old Variable Names

The proof rules we present in Chapter III replace some, but not nearly all, of Krone's [25] rules, which employ a programming language statement, **remember**. This statement appears in our rule (Figure 40) that serves as a bridge between our rules and those of Krone that we will still be using. Interpreting this **remember** statement is just the same as applying the Rem function:

$$\mathcal{I}(\mathbf{remember})(\text{env}_{\text{NL}}) \stackrel{\text{def}}{=} \text{Rem}(\text{env}_{\text{NL}}) \quad (2.49)$$

In final preparation for defining the interpretation of a **while** loop, let MFP stand for the minimum fixed point operator, let WL denote an arbitrary environment change ( $\text{WL} : \text{Environments} \rightarrow \text{Environments}$ ), and let the functional  $\Lambda_{\text{W}}$  define the next approximation of WL ( $\Lambda_{\text{W}} : (\text{Environments} \rightarrow \text{Environments}) \rightarrow (\text{Environments} \rightarrow \text{Environments})$ ):

$$\begin{aligned} \mathcal{I}(\mathbf{loop\ maintaining\ Inv\ while\ } b\_p\_e \mathbf{\ do\ SS\ end\ loop})(\text{env}_{\text{NL}}) & \quad (2.50) \\ & \stackrel{\text{def}}{=} \text{Fgt}(\text{MFP}(\Lambda_{\text{W}})(\text{Rem}(\text{env}_{\text{NL}}))) \quad \text{where} \end{aligned}$$

$$\Lambda_{\text{W}}(\text{WL})(\text{env}) = \begin{cases} \text{env} & \text{if } \text{AE}(\text{env}) \neq \text{NL} \\ [\text{CF}, \text{cs}, \text{ns}, \text{se}, \text{os}, \text{d}] & \text{else if } \neg \mathcal{I}(\text{Inv})(\text{env}) \\ \text{env} & \text{else if } \neg \mathcal{I}(b\_p\_e)(\text{env}) \\ \text{WL}(\mathcal{I}(\text{SS})(\text{env})) & \text{otherwise} \end{cases} \quad (2.51)$$

Although this definition of the interpretation of the **while** loop only applies when  $\text{AE}(\text{env}_{\text{NL}}) = \text{NL}$ , as a technical matter  $\Lambda_{\text{W}}$  must be total, i.e., we must define  $\Lambda_{\text{W}}(\text{WL})$  in all environments. So, for all environments  $\text{env}$  such that  $\text{AE}(\text{env}) \neq \text{NL}$  we define  $\Lambda_{\text{W}}(\text{WL})(\text{env}) = \text{env}$ . Otherwise, if the invariant is violated ( $\neg \mathcal{I}(\text{Inv})(\text{env})$ ), the next approximation sets the assert-status to categorically false, leaving the remaining components of the environment unchanged. If neither of the above conditions applies and the Boolean program expression in the **while** clause evaluates to false ( $\neg \mathcal{I}(b\_p\_e)(\text{env})$ ), the next approximation leaves the environment unchanged; this represents terminating loop execution. Otherwise, the next approximation is the interpretation of the current approximation in the environment that results from interpreting the body of the loop; this represents performing the first iteration of the loop, and, then, performing the remainder of loop execution.

### 2.2.3 Validity

Recall that an assertive program is the combination of an executable program and its specification. Furthermore, (see page 31) an assertive program that satisfies its specification in every possible execution is said to be *valid*. The specification *promises*

results *assuming* certain conditions hold for the environment. The purpose of the assert-status is to record, for each particular execution, whether it is satisfying the specification.

One of the assumed conditions, discussed on page 53, is that the environment contain the right meaning for procedure  $D$  when  $D$ 's declaration is encountered. If not, then the assert-status is set irrevocably to vacuously true (VT). When the interpretation of a program in a neutral environment yields a vacuously true environment, we know that some assumption has been violated; we know that this environment gives us little information about the validity of the program. The program is computing relative to a specification of the form  $H_1 \Rightarrow H_2$  in an environment where  $H_1$  is false.

Among the promised results is that the environment satisfies the precondition of procedure  $P\_nm$  when  $P\_nm$  is called. If not, then the assert-status is set irrevocably to categorically false (CF). When the interpretation of a program in a neutral environment yields a categorically false environment, we know that the program has failed to deliver some promised result; this environment has given us the important information that the program is invalid. Because VT and CF are “stuck states,” interpretation of a program in an environment that is not neutral gives us little (or no) information.

This discussion of assert-status in our denotational semantics gives rise to an equivalent, but more formal, definition of program validity: an assertive program,  $Prog$ , is *valid* if and only if the interpretation of  $Prog$  in every neutral environment yields an environment that is not categorically false. We can also state this definition using our mathematical notation:

**Definition 2.1** *Program  $Prog$  is valid if and only if, for every environment  $env$  such that  $AE(env) = NL$ ,  $AE(\mathcal{I}(Prog)(env)) \neq CF$ .*

In our introduction (Chapter I), we wrote of a program-with-specification satisfying the correctness conjecture, meaning that the program is correct with respect to its specification. In this chapter, “programs-with-specification” have become “assertive programs,” and “satisfying the correctness conjecture” has become “being valid.” We also discussed that formal bases for reasoning about the behavior of computer programs work by transforming a program-with-specification into a formula of classical mathematics. We further pointed out (page 7) that the idea “is that if the final purely mathematical formula is true, then the correctness conjecture for the original program and specification is also true.” Consistent with our change in terminology, we want to say that a true mathematical statement is “valid.” We also want to carefully define this term. Mathematical statements are expressed in a formal language belonging to a formal mathematical proof system; such a system, including its set of axioms, is often called a *theory*.

**Definition 2.2** *A mathematical statement is valid if and only if it evaluates as true in every model (see page 5) of its theory.*

We provide, in Chapter III, a purely syntactic method for establishing the validity of assertive programs. This method relies on transforming assertive programs to mathematical statements. If we have used the method to transform assertive program Prog into mathematical statement  $H$ , we will know that Prog is valid if  $H$  is valid.

